# Reliable DAG Scheduling on Grids with Rewinding and Migration

# Reliable DAG scheduling on Grids with Rewinding and Migration

Israel Hernandez and Murray Cole
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

j.i.hernandez@sms.ed.ac.uk, mic@inf.ed.ac.uk

## Abstract

Fault tolerance is an important issue in Grid Computing as the availability of Grid resources can not be guaranteed. Effective scheduling methods must include fault tolerant mechanisms to preserve the execution of DAG applications, despite the presence of a processor failure. To address this, we designed the DAG rewinding mechanism, an event-driven process executed when a failure is detected at some rescheduling point. The rewinding mechanism preserves the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks. The mechanism rewinds the progress of the application to a previous state, thereby preserving the execution despite the failed processor(s). This paper extends our work in the area by adding the rewinding mechanism to our previous dynamic scheduling methods $GTP$ and $GTP/c$. We show how to integrate the rewinding mechanism within our dynamic execution models.

**Keywords:** Fault tolerance, Grid computing, parallel processing, DAG scheduling.

## 1 Introduction

Grid systems are emerging as a distributed computational platform suitable for executing scientific applications. Such applications are often abstracted as directed acyclic graphs (DAGs), in which vertices represent application tasks and edges represent data dependencies between tasks. The core scheduling issues are that the availability and performance of grid resources, can be expected to vary *dynamically*, even during the course of an execution. Our previous dynamic models $GTP$ [1] and $GTP/c$ [2] address this issue by allowing rescheduling and migration of tasks in response to significant variations in resource characteristics. However, such dynamic models are not designed to react to processor failure during execution. Little work [4, 7] has been conducted to design fault tolerant mechanisms for DAG applications. To address this, we propose a rewinding mechanism which considers the recomputation and migration of those tasks (even if they have finished execution) whose loss would otherwise disrupt execution of succeeding tasks. We extend our work in this area, by including the rewinding mechanism into our previous scheduling methods $GTP$ [1] and $GTP/c$ [2]. The remainder of this paper is structured as follows. In Section 2 we explain how the DAG application is affected when a processor failure occurs during execution. In Section 3 we show how to integrate the rewinding mechanism into $GTP$ and in Section 4 we show the integration into $GTP/c$. Section 5 presents the results of some simulated executions using the rewinding mechanism. Section 6 describes related and future work.


*GridNets 2007*, October 17–19, 2007, Lyon, France.


## 2 Reliable DAG scheduling with rewinding

Our previous work in Grid scheduling of DAG applications is sketched in figure 1, in which we address

the dynamicity of Grids with cyclic use of a static mapping method. The first version of the mapping method, the *GTP* system [1] addresses the dynamic nature of Grids by allowing rescheduling and migration of tasks where such migration helps to reduce the earliest finish time of tasks.

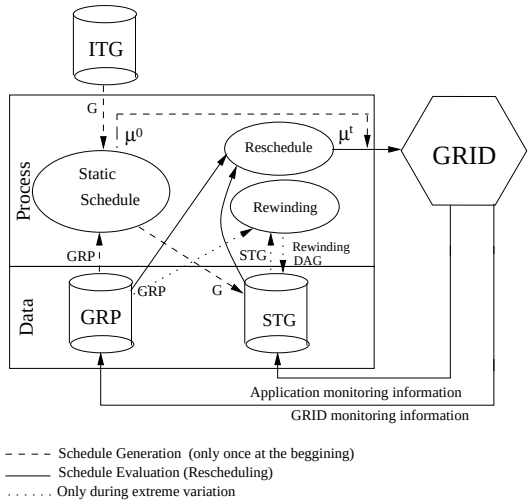


Figure 1: The GTP mapping method

The second version, the *GTP/c* system [2], is focused on reducing the impact of the migration cost on execution time (makespan) by maintaining a collection of reusable copies of the results of completed tasks, derived from the migration strategy defined for *GTP*. Though *GTP* and *GTP/c* react to dynamic changes in Grid resources, they are not able to react to extreme changes (i.e. processor failure). The presence of a resource failure in a particular processor during execution, may disrupt the subsequent execution of other tasks. The tasks expected to be disrupted can be grouped as: a) those succeeding tasks still retrieving data from preceding tasks already executed on the failed processor, and b) those unfinished tasks mapped to the failed processor which have begun to gather input data for execution. The proposed rewinding mechanism, an event-driven process executed when a processor failure is detected at some rescheduling point (RP), seeks to preserve the execution of the application by recomputing and migrating those tasks which will cause these problems. We identify three main steps to the integration of the rewinding mechanism into a particular dynamic mapping approach,

1. The first step is related to the integration of the rewinding mechanism with the data structures containing the information on both the performance of the processors composing the Grid system and the progress of the application (i.e. STG and GRP defined below).

2. The second step is related to the procedure of the rewinding mechanism itself, which will rewind those critical tasks associated with the failed processor.

3. The last step is related to particular considerations in the dynamic scheduling strategy (i.e. copying, data replication) and deals with resetting the information maintained in the system and linked to the failed processor, to avoid inconsistencies in subsequent scheduling decisions.

# 3 GTP with rewinding

We recall that our earlier GTP system allows rescheduling and migration of tasks in response to variations in the performance of Grid resources. Details of the costing of candidate schedules can be found in [1]. The inclusion of the rewinding mechanism into *GTP* produces the *GTP/r* version. As stated above, we first need to identify the data structures containing the information on both the performance of the processors composing the Grid system and the progress of the application.

## 3.1 Definition of the Grid architecture

We represent Grid Resource Pools (GRP) with graphs $GRP :: (P, L,$avail,bandwidth$)$ where $P$ is the set of available processors in the system, $p_i(1 \leq i \leq |P|)$. $L$ is the set of communication links connecting pairs of distinct processors, $l_i(1 \leq i \leq |L|)$ such that $l(m, n) \in L$ denotes a communication link between $p_m$ and $p_n$. The dynamic scheduling decisions are based upon the latest available resource

performance information (as returned by standard Grid monitoring tools such as NWS or Globus MDS). Thus, at time $t$ we assume knowledge of $avail^t :: P \rightarrow [0..1]$, capturing the availability of each CPU and $bandwidth^t :: L \rightarrow Float$ capturing the available bandwidth on each link. We assume that the intra-processor communication cost ($p_m = p_n$) is negligible. Failures in traditional distributed systems [3] are mostly linked to physical failures which make the resources unavailable. However, in a Grid context, a failure embraces other situations, which affect the availability of resources. For instance, in the context of the DAG execution, a particular processor might be assigned to another job with higher priority in a manner outside our control.

### 3.2 Definition of ITG structure

Static information about the DAG application is represented by an *input task graph* $ITG :: (V, E, data, W)$. $V$ is the set of tasks, $v_i (1 \leq i \leq |V|)$. $E \subseteq V \times V$ is the set of directed edges connecting pairs of distinct tasks, $e_i (1 \leq i \leq |E|)$, where $e(i, j) \in E$ denotes a direct dependency and data transfer from task $v_i$ to $v_j$. For future convenience, we define $Pred(v_i)$ to denote the subset of tasks which directly precede $v_i$ and $Succ(v_i)$ to denote the subset of tasks which directly succeed $v_i$. $Level(v_i)$ denotes how deep in terms of number of edges, a task $v_i$ is from the entry node. We assume that information about data transfer sizes and task computation times are provided in standard units, compatible with those of our bandwidth and computational performance measures. We use $data :: V \times V \rightarrow Int$ to describe the size of data transfers, such that $data(i, j)$ denotes the amount of data transfer from $v_i$ to $v_j$. Remembering that our processors are heterogeneous, we represent computation times with $W :: V \times P \rightarrow Int$, where $W(i, m)$ denotes the execution time in standard units of task $v_i$ on processor $p_m$.

### 3.3 Definition of the STG Structure

We maintain additional dynamic information on the progress of the tasks. We model this by augmenting the static ITG, to form a *Situated Task Graph* $STG$. This includes information on current schedule of tasks and partial completion of both tasks and communications. This is necessary, together with monitored information on the availability of processors and links, to allow $GTP$ to iteratively compute improved schedules. We define $STG :: (V, E, data, W, \Pi, \kappa^c, \kappa^d)$, where the first four components are taken directly from the corresponding $ITG$. We use $\Pi :: V \rightarrow P^+$ to represent placement information. $P^+$ represents $P$ augmented with the special value $NONE$. $GTP$ includes the concept of *placed task*. A task is said to become *placed* on a processor once it has begun to gather its input data on that processor. Otherwise, it is considered as *non-placed*. The distinction is important because of its impact on migration costs associated with data retransmission. The decision to migrate a non-placed task will incur no additional migration cost because retransmission of data is not needed. For placed tasks $v_i$, $\Pi(v_i)$ indicates the corresponding processor. For non-placed tasks $v_i$, $\Pi(v_i) = NONE$. A placed task remains placed until migrated or until the whole application terminates, because even after task completion we may later need to retrieve its results. For future convenience, we define $Q^t :: P \rightarrow \mathcal{P}(V)$ to denote the subset of placed tasks mapped on each $p_i \in P$ at $t$. $GTP$ assumes that information concerning the progress of computations and communications is made available by monitoring mechanisms at each rescheduling point. We use $\kappa^c :: V \rightarrow [0..1]$ to capture the proportion of a task's computation which has been completed, and similarly, $\kappa^d :: E \rightarrow [0..1]$ to capture the proportion of a data transfer which has been completed.

### 3.4 Procedure of the $GTP/r$ system

In this section we integrate the rewinding mechanism into the $GTP$ model. To rewind a task $v_i$, at time $t$, we must perform the following operations on the $STG$ data structure.

1. $\forall v_j \in SUCC(v_i)$ *set* $\kappa^d(v_i, v_j)$ *to* 0
2. $\forall v_k \in PRED(v_i)$ *set* $\kappa^d(v_k, v_i)$ *to* 0
3. *Set* $\kappa^c(v_i)$ *to* 0

4. *Set* $\Pi(v_i)$ *to* $NONE$

Thus, assuming that $p_m$ is the failed processor, we have that $Q^t(p_m) = \{v_0, v_1, v_2, ..., v_k\}$ contains the set of $k$ placed tasks known at time $t$ mapped onto $p_m$, from which we will rewind those placed tasks which are expected to disrupt the forward execution of succeeding tasks. To do this, we must consider each task in $v_i \in Q^t(p_m)$. Intuitively, $v_i$ must be rewound if either, i) it has a successor task which has not yet received a complete copy of the result of $v_i$, or ii) it has a successor $v_j$, which is also assigned to $p_m$ and which also needs to be rewound.

The recursive form of this rule means that we must consider tasks in $Q^t$ in an order which respects a reverse topological sort (according to the precedence constraints between tasks). Thus, within $Q^t(p_m)$ we must consider exit tasks first, then their predecessors, and so on. This ordering is straightforward to maintain in an implementation because all precedence information is available. Thus, a task $v_i \in Q^t(p_m)$ must be rewound if,

1. $\exists e(v_i, v_j) \in E : \kappa^d(v_i, v_j) < 1$, *or*
2. $\exists v_k \in SUCC(v_i) : v_k \in Q^t(p_m)$ *and* $v_k$ *must be rewound*

Note the importance of maintaining information about all placed tasks in $Q^t$, including those whose completion is complete. Following the procedure, we now know that no information related to the failed processor $p_m$ is maintained in $GTP/r$. Obviously, after the rewinding process, the failed processor will not be considered in the subsequent scheduling decisions, unless $avail^t(p_m) > 0$ in future RP's.

To illustrate the rewinding mechanism, we will use the example of figure 2(a) where we observe a processor failure in $p_3$ at some point between $RP_{n+1}$ and $RP_{n+2}$. We observe that such failure will inhibit the precedence constraint satisfaction for $e(v_2, v_3)$ as $v_3$ will stop retrieving the input required from $v_2$ to start execution. The failure will be detected at $RP_{n+2}$ and the rewinding mechanism will be triggered. The rewinding mechanism must determine which placed tasks mapped to $p_3$ need to rewind to preserve the execution of the DAG application. At $RP_{n+2}$, $Q^{n+2}(p1) = \{v1\}$, $Q^{n+2}(p3) = \{v0, v2\}$ and $Q^{n+2}(p4) = \{v3\}$. The rewinding mechanism will evaluate in reverse order the sequence of placed tasks $v_i \in Q^{n+2}(p3)$. Thus, the first task to evaluate is $v_2$ which inhibits the precedence constraint satisfaction for $e(v_2, v_3)$, as $v_3$ will stop retrieving input from $v_2$ executed on $p_3$. Thus, $v_2$ is rewound following the form explained above. Now, the next task to evaluate from $Q^{n+2}(p3)$ is $v_0$, which $Succ(v_0) = \{v1, v2\}$, and the first precedence constraint $e(v0, v1)$ is satisfied as $v_1$ has finished their execution at $p_1$. However, the second precedence constraint $e(v0, v2)$ will not be satisfied as $v_2$ (already rewound) will not be able to retrieve their input from $v_0$ executed on $p_3$. Thus, task $v_0$ must also be rewound. Since, task $v_0$ and $v_2$ were rewound, they will be ready to be rescheduled and migrated to a different available processor. Following the steps for the rewinding mechanism, there is no additional information linked to $p_3$ which could lead to inconsistencies in scheduling decisions. After rewinding and rescheduling the application at $RP_{n+2}$, the task $v_3$ was finally executed at $p_4$ after receiving the required inputs.

# 4 GTP/c with Rewinding

We will similarly follow the three steps outlined to integrate the rewinding mechanism into the $GTP/c$ system to produce the $GTP/c/r$ version.

## 4.1 Definition of GRP and STG

Our definition of GRP (Grid Resource Pools) and ITG (Initial Task Graph) are identical to those from $GTP$ defined in section 3.1 and 3.2 respectively. We recall that $GTP/c$ now maintains a collection of reusable copies to reduce the impact of the migration cost on makespan. Thus, $GTP/c$ includes in the $STG$ structure, $\Omega :: E \rightarrow \mathcal{P}(P)$ to capture information on location of copies. Details of the costing of candidate schedules can be found in [2].

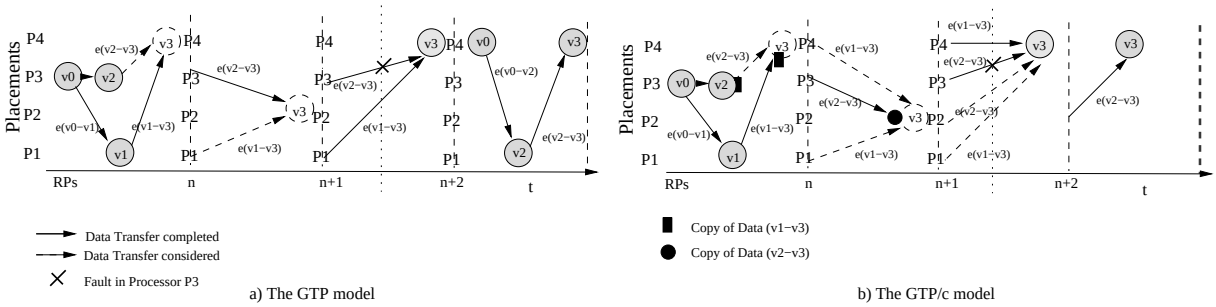


Figure 2: The rewinding mechanism for $GTP$ and $GTP/c$

### 4.2 Procedure of the GTP/c/r system

The rewinding mechanism for $GTP/c/r$ now includes a criterion related to the existence of possibly reusable copies in $\Omega(e_{i,j})$ for a particular edge $e(v_i, v_j) \in E$. We consider that if there exist at least one reusable copy in a processor different than $p_m$, then it means that $v_j$ can retrieve the data from this copy despite $p_m$'s failure, and therefore that rewinding is not needed. This particular feature of $GTP/c$ is expected to reduce the overhead cost generated by the rewinding mechanism. For $GTP/c/r$, a task $v_i \in Q^t(p_m)$ must be rewound if,

1. $\Omega(v_i) = \{p_m\}$, *(this is the only copy), and either*
2. $\exists (v_i, v_j) \in E : \kappa^d(v_i, v_j) \leq 1$, *or*
3. $\exists v_k \in SUCC(v_i) : v_k \in Q^t(p_m)$ *and* $v_k$ *must be rewound*

As before, for tasks to be rewound, we must reset elements of $\kappa^d$, $\kappa^c$ and $\Pi$ to reflect the rewinding.

For $GTP/c/r$, all the copies located at the failed processor $p_m$ and maintained in $STG$ can lead to scheduling thrashing if they are not eliminated. Thus, as stated in the last step of the procedure, those copies $\Omega^t(e_{i,j}) = p_m$ must be eliminated from $STG$. To illustrate $GTP/c/r$, we will use the same example from $GTP/r$, with failure in processor $p3$ at some point between $RP_{n+1}$ and $RP_{n+2}$. This is shown in figure 2(b). At $RP_{n+2}$, $Q^{n+2}(p1) = \{v1\}$, $Q^{n+2}(p3) = \{v0, v2\}$ and $Q^{n+2}(p4) = \{v3\}$. The first task to evaluate is $v_2$ which, as we observe, inhibits the precedence constraint satisfaction for $e(v_2, v_3)$, as $v_3$ will stop retrieving input from $v_2$ executed on $p_3$. However, due to the maintenance of reusable copies for $GTP/c/r$, the input required by $v_3$ from $v_2$ can be retrieved from the copy stored at $p_2$, satisfying the precedence constraint. Thus, rewinding task $v_2$ is not needed. The next task to be evaluated is $v_0$ with $Succ(v_0) = \{v1, v2\}$. The first precedence constraint for $e(v_0, v_1)$ is satisfied as $v_1$ has finished execution at $p_1$. The next precedence constraint for $e(v_0, v_2)$ is considered to be satisfied as $v_2$ kept its status of finished task, because it was not rewound. Thus task $v_0$ will not be rewound. Finally, since $GTP/c/r$ maintains a collection of reusable copies some of which may be stored at $p_3$, we need to delete those copies stored at $p_3$ to avoid inconsistency in future decisions. In this case we delete the copy $\Omega_l(v_2, v_3)$ as it could lead to subsequent inconsistency in scheduling decisions if task $v_3$ was migrated in the future. Thus, after the third step, the application has been rewound. Completing the example, after rewinding and rescheduling the application at $RP_{n+2}$, $v_3$ is finally executed at $p_4$ after receiving the required inputs.

## 5 Performance Evaluation

Our evaluation is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. We have used the Simgrid simulator [5] for this purpose. We evaluated the versions $GTP/r$ and $GPT/c/r$ which include the DAG

rewinding mechanism.

## 5.1 Comparison Metrics

We use the *Normalized schedule length (NSL)* to determine the amount of extra time that a particular application requires to finish execution when processor failure. The NSL metric is defined as the ratio of the schedule length (makespan) to the sum of the computational weights along the critical path and can be computed as

$$NSL = \frac{Makespan}{\sum_{v_i \in CPath} \overline{W_i}}$$

Thus, the amount of extra time required (AET) can be determined by $AET = NSL_r - NSL$, corresponding to difference between the $NSL_r$ (application using rewinding mechanism) value and the $NSL$ value (application without rewinding).

Other complementary metrics to consider are the Rewound Tasks (RT) metric to determine the number of placed tasks rewound, the rewinding overhead to determine the overhead incurred and the Levels Rewound (LR) metric to determine how deep the application had to be rewound in the presence the failed processor.

## 5.2 DAG applications

The shape of the DAGs considered in our experiments were taken from the Standard Task Graph Project (STDGP) [6]. Since $STDGP$ considers the DAGs to be executed in a homogeneous environment, without communication cost, we had to add randomly (but repeatably) generated $W$ and *data* information to produce our ITGs. The graph size (in number of tasks) varied in the range {50,100,300,500,1000} and the average number of edges per graph-size varies in the range {300,800,100,10000,35000} respectively. In keeping with the principles of schedule feedback, we assume the availability of the latest makespan of the application, and we set the fixed-period rescheduling cycle at 10% of the value of the makespan.

## 5.3 Simulation Results

We created two different groups (TE1 and TE2) of test scenarios to evaluate the performance of the rewinding mechanism. Both groups keep the principles of our previous work in [1, 2] by involving a sequence of randomly defined (but repeatable) events, each simulating a resource change in either processor or bandwidth availability. The key difference is that we injected in TE2 an additional event simulating a processor failure to occur at the mid-point of the execution. Obviously, $GTP/r$ and $GTP/c/r$ will use TE2 to evaluate the rewinding mechanism, while $GTP$ and $GTP/c$ will use TE1 as they are not able to react to processor failure. Our scenarios are distinguished by the bound placed on the maximum variation allowed in one event, expressed as a percentage of the peak performance of a resource. For example, in the scenario with a bound of 30%, any one event can cause the availability of a processor to decrease to no less than 70% of its peak performance, or of a link to decrease to no less than 70% of its maximum bandwidth. We experimented with a bound ranging from 0% to 90% in increments of 10%. Our graphics embraces the whole spectrum of bounds. For reasons of space we report here only the results for SCE5-500 (5 Processors and 500 tasks) and SCE20-1000 (20 Processors and 1000 tasks).
The experimental results show that for most cases the performance of the rewinding mechanism for $GTP/c/r$ outperforms $GTP/r$ in the presence of a processor failure. For scenario SCE5-500 (5 PE's,500 Tasks), figure 3 shows that $GTP/c/r$ needed up to 13% of extra time compared with $GTP/c$ to executed the application in the presence of failure, and $GTP/r$ needed up to 16% of extra time compared with $GTP$. The average number of rewound levels (LR) for $GTP/c/r$ is up to 40 levels and up to 42 levels for $GTP/r$. The average of the number of rewound tasks for $GTP/c/r$ is up to 20 and up to 27 for $GTP/r$. For SCE20-1000 (20 PE's, 1000 Tasks), $GTP/c/r$ needed up to 12% of extra time compared with $GTP/c$ to execute the application in the presence of failures and $GTP/r$ needed up to 14% of extra time compared with $GTP$. The average number of rewound levels is up to 90 for $GTP/c/r$ and up

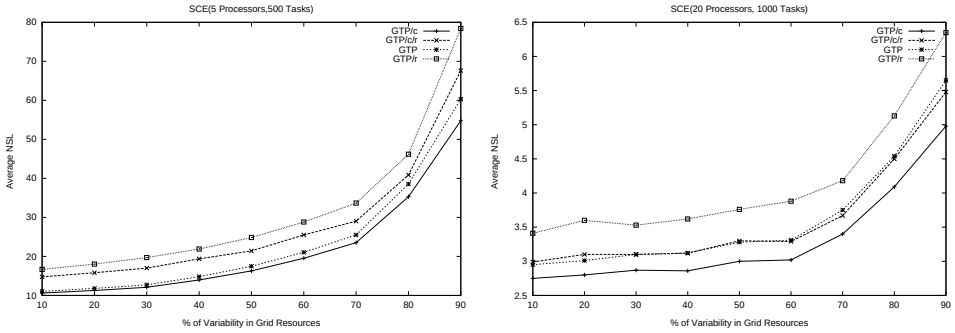


Figure 3: Average NSL for GTP/r and GTP/c/r

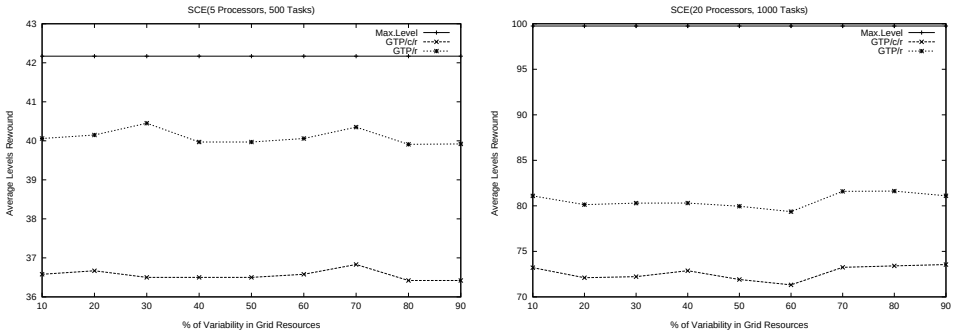


Figure 4: Average Levels Rewound for GTP/r and GTP/c/r

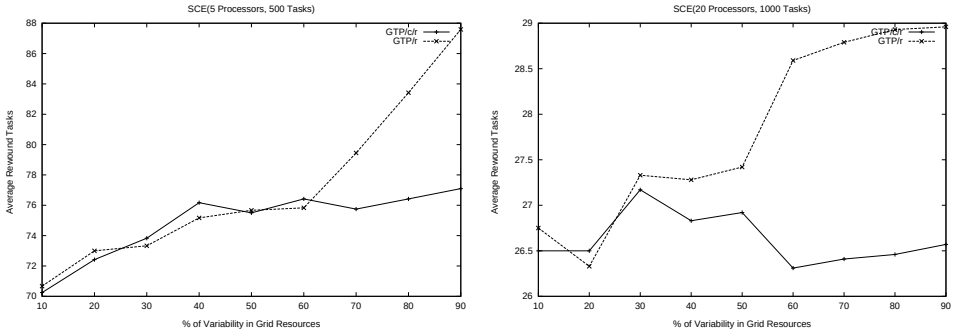


Figure 5: Average Rewound Tasks for GTP/r and GTP/c/r

to 93 and for $GTP/r$. The average of the number of rewound tasks is up to 19 for $GTP/c/r$ and up to 22 for $GTP/r$. The performance of the rewinding mechanism for a particular scheduling method, is highly dependent upon the details of the scheduling method used. Moreover, from the experiments, we observe that there exists a vicious circle linking the number of rewound levels to the number of rewound

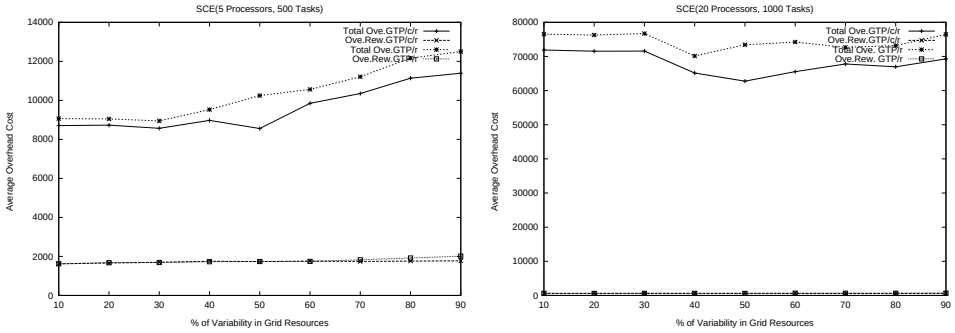


Figure 6: Average Overhead Cost for GTP/r and GTP/c/r

tasks, to the rewinding overhead cost and finally to the makespan of the application. The copying feature in $GTP/c/r$ tends to reduce the impact of the vicious circle on makespan compared with $GTP$.

# 6 Related and Future Work

Little work [4, 7] has been conducted to design fault tolerant mechanisms for DAG applications. The Directed Acyclic Graph Manager (DAGMan) [7] is a meta-scheduler for Condor jobs, which consider the submission of DAGs tasks on Grids. DAGMan contains a mechanism to tolerate faults in the software, caused mainly by human mistake (i.e. an error in an output file which is input for a succeeding task). Such a mechanism consists of the resubmission of uncompleted portions of a DAG when one or more tasks resulted in failure. If any task in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made due to the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG (input file), containing information about the progress of the DAG (unfinished and successfully finished tasks). Then, using this Rescue DAG as input file, the unfinished tasks are resubmitted. The tasks successfully completed will not be re-executed. We note that for this case, recomputation of tasks which had already finished is not needed as it assumes full availability of processors during execution.

We believe that our rewinding mechanism can also be applied in other aspects of the DAG scheduling problem. For instance, DAG schedulers usually tend to obtain a schedule of unfinished tasks, focused on minimizing the makespan. However, there could be some cases in which rewinding the DAG (recomputation of finished tasks) could derive a better makespan, even without processor failure.

# References

[1] I. Hernandez and Murray Cole, *"Reactive Grid Scheduling of DAG Applications"*, Proceedings of 25th IASTED (PDCN), 92-97, Acta Press, 2007.

[2] I. Hernandez and Murray Cole, *"Scheduling DAGs on Grids with Copying and Migration"*, to appear in Parallel Processing and Applied Mathematics (PPAM07), Springer LNCS, 2007.

[3] Pankaj Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[4] Medeiros, Cirne, Brasileiro and Sauve *"Faults in Grids: Why are they so bad and what can be done about it?"*, Proc. 4th Int. Workshop on Grid Computing, 18-24, IEEE Computer Society, 2003.

[5] The Simgrid project, http://simgrid.gforge.inria.fr/

[6] The Standard Graph Project, http://www.kasahara.elec.waseda.ac.jp/schedule/

[7] Condor DAGMan Applications, http://www.cs.wisc.edu/condor/manual/